Pipelining

Forecast
- Big Picture
- Datapath
- Control
- Data Hazards
  - Stalls
  - Forwarding
- Control Hazards
- Exceptions

Motivation

Want to minimize:
- Time = Instructions/\text{prog} \times \text{CPI} \times \text{Cycle time} = P \times ? \times ?

Single cycle implementation:
- CPI = 1
- Cycle=imem+reg\_rd+alu+dmem+reg\_wr+muxes & control
- = 500 + 250 + 500 + 500 + 250 + 0 + 0 = 2000 ps = 2 ns
- Time/prog = P \times 2 ns

Big Picture

Multicycle implementation:
- CPI = 3, 4, 5
- Cycle=Max(memory, registers, ALU, muxes&control)
- = max(500, 250, 500) = 500 ps
- Time/prog = P \times 4 \times 500 = P \times 2000 ps = P \times 2 ns

Would like:
- CPI = 1 + hazards
- Cycle = 500 ps + overheads
- In reality, \sim 3x improvement
Big Picture

Instruction Latency = 5 cycles
Instruction Throughput = 1/5 instructions per cycle
CPI = 5 cycles per instruction

Pipelining: process instructions like a lunch buffet!
ALL microprocessors today employ pipelining for speed
E.g., Intel PentiumIII and Compaq Alpha 21264

But
• datapath? note: five instructions in datapath in cycle 5
• control? must be generated by multiple instructions
• instructions may have data and control flow dependences
**Datapath (Fig. 6.11)**

Program execution order (in instructions):

- **lw $1, 100($0)**
- **lw $2, 200($0)**
- **lw $3, 300($0)**

**Time (in clock cycles):**

**CC 1** | **CC 2** | **CC 3** | **CC 4** | **CC 5** | **CC 6** | **CC 7**
---|---|---|---|---|---|---
lw $1, 100($0) | ALU | Reg | DM | Reg |
lw $2, 200($0) | ALU | Reg | DM | Reg |
lw $3, 300($0) | ALU | Reg | DM | Reg |

---

**Datapath (Fig. 6.10)**

**IF: Instruction** | **Mux** | **EX: Execute/ register file read** | **MEM: Memory access** | **WB: Write back**
---|---|---|---

**Control**
- Set by five different instructions
- Divide and conquer: carry IR down the pipeline

**MIPS ISA requires the appearance of sequential execution**

---

**Data Dependence**

One instruction produces a value used by a later instruction

E.g.,

- **add $1, -, -**
- **sub -, $4, -**

```
<table>
<thead>
<tr>
<th>i</th>
<th>F</th>
<th>D</th>
<th>X</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+1</td>
<td>F</td>
<td>D*</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>
```
Data Dependence

Simple solution: Stall the pipeline

E.g.,
- add $1, -, -
- sub -, $4, -

1 2 3 4 5 6 7 8 9

add F D X M W*
sub F D* X M W

But CPI > 1, we will do better using “register forwarding”

Control Dependence

One instruction affects which instruction will execute next

E.g., bne, j
- sw $4, 0($5)
- bne $2, $3, loop
- sub -, -, -

1 2 3 4 5 6 7 8 9

sw F D X M W
bne F D X* M W
sub F D X M W

CPI > 1, we will do better

Pipelined Datapath

Single-cycle datapath (Recall Fig. 6.10)

Pipelined execution
- assume each instruction has its own datapath (Fig. 6.11)
- but each instruction uses different part in every cycle
- multiplex all on one datapath
- latch to separate cycles (as in multicycle) and instructions!

Ignore data and control flow dependences for now
- data hazards
- control flow hazards
Pipelined Datapath (Fig. 6.12)

Instruction flow
- add and load
- write of registers
- pass register specifiers

Any info needed by a later stage will be passed down
- store value through EX

Pipelined Control

IF and ID
- none

EX
- ALUop, ALUsrc, Regdst

MEM
- Branch MemRead, MemWrite

WB
- MemtoReg, RegWrite

Figure 6.25
Pipelined Control

But controlled by different instructions

Decode instructions and pass the signals down the pipe

Control sequencing is embedded in the pipeline

Pipelining

Not too complex yet

• data hazards
• control hazards
• exceptions
Data Hazards

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)

Data Hazards

Must first detect hazards

ID/EX.WriteRegister = IF/ID.ReadRegister1
ID/EX.WriteRegister = IF/ID.ReadRegister2
EX/MEM.WriteRegister = IF/ID.ReadRegister1
EX/MEM.WriteRegister = IF/ID.ReadRegister2
MEM/WB.WriteRegister = IF/ID.ReadRegister1
MEM/WB.WriteRegister = IF/ID.ReadRegister2

Data Hazards

Not all hazards because some
- WriteRegister not used e.g., sw
- ReadRegister not used e.g., addi, jump
- Do something only if necessary

Data Hazards

Hazard detection unit
- several 5-bit (or 6-bit) comparators

Response? Stall pipeline
- Instructions in IF and ID stay
- IF/ID pipeline latch not updated
- send “nop” down pipeline - called a “bubble”
- PcWrite, IF/IDWrite and nop mux
Register Forwarding (Figure 6.38)

Data Hazard

A better response - forwarding
all of the above made sure reg read after reg write
Instead of stalling
  • use mux to select forwarded value rather than reg value
  • control mux with hazard detection logic

Data Hazards

Load followed by a use
Can't avoid a stall
Stall one cycle and the forward

Other options
Disallow hazardous sequences
  • compiler will never generate them
  • assembly programmers will not use them
  • If used, result is random
Control Flow Hazards

Control flow instructions
- branches, jumps, jals, returns
- can’t fetch until branch outcome known
- too late for next IF

What to do?
- Always stall
- easy to implement
- performs poorly
- 1/6th instructions is a branch, each branch takes 3 cycle
- what is the CPI?

Predict branch not taken
let sequential instructions go down the pipeline
must kill later instructions if incorrect
must stop memory accesses and reg writes
  • including loads (why?)

Late flush of instructions on misprediction
Complex
Control Flow Hazards

Even better but more complex
  • predict taken
  • predict both
  • dynamically adapt to program branch patterns
  • significant fraction of chip real estate
    • PentiumIII
    • Alpha 21264
  • current topic of research

Another option: delayed branches
  • always execute following instruction
  • delay slot
  • put useful instruction, nop otherwise

losing popularity

Exceptions

add $1, $2, $3 overflows!

a surprise branch
  • earlier instruction flow to completion
  • kill later instructions
  • save PC in EPC, PC to exception handler, Cause, etc

cost a lot of designer sanity

Even worse: in one cycle
  • I/O interrupt
  • user trap to OS
  • illegal instruction
  • arithmetic overflow
  • hardware error
  • etc
State of the Art: Superscalar

<table>
<thead>
<tr>
<th>i</th>
<th>F</th>
<th>D</th>
<th>X</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+5</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>i+7</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

IF: parallel access to I-cache, require alignment?
ID: replicate logic, fixed length instrs? hazard checks? dynamic?
EX: parallel/pipelined
MEM: >1 per cycle? If so, hazards, multi-ported register D-cache?
WB: different register files? multi-ported register files?
  more things replicated
  more possibilities for hazards
  more loss due to hazards (why?)

State of the Art: Out of Order

• execute later instructions while previous is waiting
• decouple into different units
• one to fetch/decode, several to execute, one to write back
• fetch in program order
• execute out of order speculatively!
• commit in order

Out of Order in the Limit

![Execution Wavefront Diagram]

- Program Form
- Processing Phase
- instruction fetch & branch prediction
- dependence checking & dispatch
- instruction issue
- instruction execution
- instruction reorder & commit
- completed instructions
A Generic Out of Order Processor

Review

Big picture
Datapath
Control
  - data hazards
  - stalls
  - forwarding
  - control flow hazards
  - branch prediction

Exceptions